10 Javadocs, Testing Exception Use Mutating ArrayList, Loops

Goals

The first part of the lab you will learn how to generate *Javadoc* documentation, how to test whether the program throws a correct exception with a correct message, and practice reading *Javadoc* style documentation for programs.

The second part introduces *ArrayList* class from the **Java Collections Framework** library, lets you practice designing methods that mutate *ArrayList* objects. We will continue to use the generics (type parameters), but will do so by example, rather than through explanation of the specific details.

In the third part of the lab you will learn how to how to convert recursive loops to imperative (mutating) loops using either the **Java** *while* statement of the **Java** *for* statements to implement the imperative loops.

10.1 Documentation, Testing exceptions, Java Libraries

For this lab download the files in *Lab10-Sp2008.zip*. The folder contains the following files:

- The file *Balloon.java* — out sample data class
- The file *ISelect.java* — the interface for a generic predicate method
- The files *RedBallon* and *SmallBalloon* that implement the *ISelect* interface for the *Balloon* data.
- The files *IList.java*, *MTList.java*, and *ConsList.java* that define a generic cons-list that implements the *Traversal* interface.
- The file *ArrListTraversal.java* shows how we can define a *Traversal* wrapper for the *ArrayList* class.
- The file *TopThree.java* will be used to practice working with *ArrayList* in imperative style (using mutation).
- The *Algorithms.java* file shows an implementation of several algorithms that consume data generated by a *Traversal* iterator and illustrates a number of ways in which loops can be implemented in *Java*.

- The *Examples.java* file that defines examples of all data and defines all tests.

Create a new **Project** *Lab10* and import into it all files from the zip file. Import the tester.jar and colors.jar.

Generating Documentation

- Once Eclipse shows you that there are no errors in your files select **Generate Javadoc...** from the **Project** pull-down menu. Select to generate docs for all files in your project with the destination *Lab10/doc* directory. Make sure you select all files for which you wish to generate the documentation.

  You should be able to open the *index.html* file in the *Lab10/doc* directory and see the documentation for this project. Compare the documentation for the class *ConsList* with the web pages. You see that all comments from the source file have been converted to the web document.

  Observe the format of the comments, especially the /** at the beginning of the comment. If you do not understand the rules, ask the TA or one of the tutors, or experiment with new comments. From now on all of your work should have a proper Javadoc style documentation.

- Now use the documentation to see what are the fields in various classes and what methods have been defined already.

- The handout shows you the relationship between all these classes and interfaces.

- Define a method *isHit* in the class *Balloon* that determines whether a shot aimed at the given *x* and *y* coordinate hits this *Balloon*. Add documentation in the *Javadoc* style. Of course, add tests in the *Examples* class. Run the tests, then rebuild the *Javadoc*s and make sure your documentation shows up correctly.

Defining and Handling Exceptions

- The files *IList.java*, *MTList.java*, and *ConsList.java* illustrate how methods can *throw* exceptions when something goes wrong.

The method *contains* in the class *Algorithms* illustrates how the *contains* method handles the exceptions that may be thrown when invoking one of the *Traversal* methods.

Can you construct an example for the method *contains* to the *Examples* class that will cause the exception to be thrown?

- The *tester* allows the programmer to test whether a method invocation by a given instance with the given list of arguments throws the expected exception and produces the expected message.

  The test case header is:

```
checkExpect (T object,
             java.lang.String method,
             java.lang.Object[] args,
             java.lang.Exception e)
```

  and a sample of its use is:

```
  // invoke getFirst() on the instance of MTList
  // should throw IllegalUseOfTraversalException
  // and produce a message "No first element in an empty list"
  public void testExceptions(Tester t){
    t.checkExpect(new MTList<Object>(),
        "getFirst",
        new Object[0],
        new IllegalUseOfTraversalException(
            "No first element in an empty list"));
}
```

  Add to the *Examples* this test and add one more test that will make sure that the method *getRest* when invoked on an instance of *MTList* also throws an exception. See what happens when you provide an incorrect *Exception* class or an incorrect message in your test case. See what happens when the method does not throw any expected exception.

ArrayList and Java Libraries

- The class *TopThree* now stores the values of the three elements in an *ArrayList*. Complete the definition of the *reorder* method. Use the previous two parts as a model. Look up the documentation for the Java class *ArrayList* to understand what methods you can use.

  Do not forget to run your tests.

10.2 Using ArrayLists and Traversals

Using ArrayList with Mutation

In this part of the lab we will work on lists of balloons.

Open the web site that shows the documentation for Java libraries

http://java.sun.com/j2se/1.5.0/docs/api/.

Find the documentation for *ArrayList*.

Here are some of the methods defined in the class *ArrayList*:

```
// how many items are in the collection
int size();

// add the given object of the type E at the end of this collection
// false if no space is available
boolean add(E obj);

// return the object of the type E at the given index
E get(int index);

// replace the object of the type E at the given index
// with the given element
// produce the element that was at the given index before this change
E set(int index, E obj);
```

Other methods of this class are *isEmpty* (checks whether we have added any elements to the *ArrayList*), *contains* (checks if a given element exists in the *ArrayList* — using the *equals* method), *set* (mutate the element of the list at a specific position). Notice that, in order to use an *ArrayList*, we have to add

```
import java.util.ArrayList;
```

at the beginning of our class file.

The methods you design here should be added to the *Examples* class, together with all the necessary tests.

- Design the method that determines whether the radius of the balloon at the given position in the given *ArrayList* of *Balloon*s is smaller than the given limit.
- Design the method that determines whether the balloon at the given position in the given *ArrayList* of *Balloon*s has the same size and location as the given *Balloon*.

- Design the method that increases the radius of a *Balloon* at the given index by 5.
- Design the method that swaps the elements of the given *ArrayList* at the two given positions.

10.3 Converting Recursive Loops into Imperative while Loops

- Work with the Lab handout. The first page gives you an overview of all classes and interfaces (except the *TopThree*) and the relationship between them. We introduce a dotted line from a method that consumes an instance of some class to that class.
- Read first the code for the *contains* method and for the *countSuch* method in the *Algorithms* class. These have been designed in the *classical* HtDP style.
- We will look together at the next two examples of *orMap* in the *Algorithms* class.

  We first write down the template for the case we already know — the one where the loop uses the *Traversal* iterator. As we have done in class, we start by converting the recursive method into a form that uses the accumulator to keep track of the knowledge we already have, and passes that information to the next recursive invocation.

  Read carefully the *Template Analysis* and make sure you understand the meaning of all parts.

```
TEMPLATE - ANALYSIS:
--------------------
return-type method-name(Traversal tr){
                         +--------------------+
// invoke the methodAcc: | acc <-- BASE-VALUE |
                         +--------------------+
   method-name-acc(Traversal tr, BASE-VALUE);
 }

 return-type method-name-acc(Traversal tr, return-type acc)
 ... tr.isEmpty() ...                   -- boolean     ::PREDICATE
 if true:
 ... acc                                -- return-type ::BASE-VALUE
 if false:
    +---------------+
 ...| tr.getFirst() | ...               -- E           ::CURRENT
    +---------------+

 ... update(T, return-type)             -- return-type ::UPDATE
          +---------------------------+
 i.e.: ...| update(tr.getFirst(), acc) | ...
          +---------------------------+
     +--------------+
 ... | tr.getRest() |                   -- Traversal<T> ::ADVANCE
     +--------------+

 ... method-name(tr.getRest(), return-type)  -- return-type
 i.e.: ... method-name-acc(tr.getRest(), update(tr.getFirst(), acc))
```

Based on this analysis, we can now design a template for the entire problem — with the solution divided into three methods as follows:

```
 COMPLETE METHOD TEMPLATE:
 -------------------------
 <T> return-type method-name(Traversal<T> tr){
                                +------------+
   method-name-acc(Traversal tr,| BASE-VALUE |);
                                +------------+
 }

 <T> return-type method-name(Traversal<T> tr, return-type acc){
       +--------------+
   if (| tr.isEmpty() |)
       +--------------+
    return acc;
  else
                          +--------------+
    return method-name-acc(| tr.getRest() |,
                          +--------------+
                          +---------------------------+
                          | update(tr.getFirst(), acc) |);
                          +---------------------------+
  }

  <T> return-type update(T t, return-type acc){ ...
  }
```

Task 3:

- Look at the first two variants of the *orMap* method (the recursively defined variant and the variant that uses the *while* loop. Identify the four parts (BASE-VALUE, Termination/Continuation PREDICATE, UPDATE, and ADVANCE) in each of them.

  Look also at the tests in the *Examples* class.

- After you understand how the *while* loop works, design two variants of the method that produces a new *ArrayList* that contains all elements of the original list that satisfy the given *ISelect* predicate.

  Test the methods by producing all red balloons or all small balloons.

- Design and test two variants of the andMap method that determines whether all elements of a given list satisfy the given *ISelect* predicate.

  Test the methods by checking whether a list contains all red balloons or all small balloons.

Converting while loops into for loops

If you have the time left, repeat all the parts of **Task 3** with the remaining two variants of the *orMap* — namely the one that uses the *for* loop with the *Traversal* and the one that uses *counted for* loop.